Require SSE3 for Chrome on x86

This Document is Public

Authors: pwnall@chromium.org, vapier@chromium.org, brucedawson@chromium.org, grt@chromium.org, rsesek@chromium.org, davidben@chromium.org, markchang@chromium.org, groby@chormium.org, laforge@chromium.org, thakis@chromium.org

September 2020

One-page overview

Summary

Assume SSE3 support in x86 Chrome builds. On x86 processors without SSE3 support, running Chrome will result in a crash. The Chrome installer will attempt to exit early, but might also crash.

SSE3 is easily confused with SSSE3. SSSE3 is a microarchitecture bump above SSE3, and below SSE4.

Platforms

Windows, Linux, Chrome OS, Android, Android WebView, iOS
(Code generation will be impacted on all platforms except for Mac.)

Team

pwnall@chromium.org

Bug

https://crbug.com/1123353

Code affected

//build/config/
//base/cpu_unittest.cc
//chrome/installer/

Indirectly, everything.

Design

Change Outline

Chrome currently requires SSE2 support on x86 processors.

Requiring SSE3

The code change is very straightforward, and is prototyped in https://crrev.com/c/2311044.

- Add the `-msse3` Clang option to 32-bit and 64-bit x86 builds.
- Update the Chrome installer to exit early on x86 CPUs without SSE3 support.
- Update a unit test to unconditionally execute SSE3 code.

The following support pages will need to be updated to reflect the new SSE3 requirement.

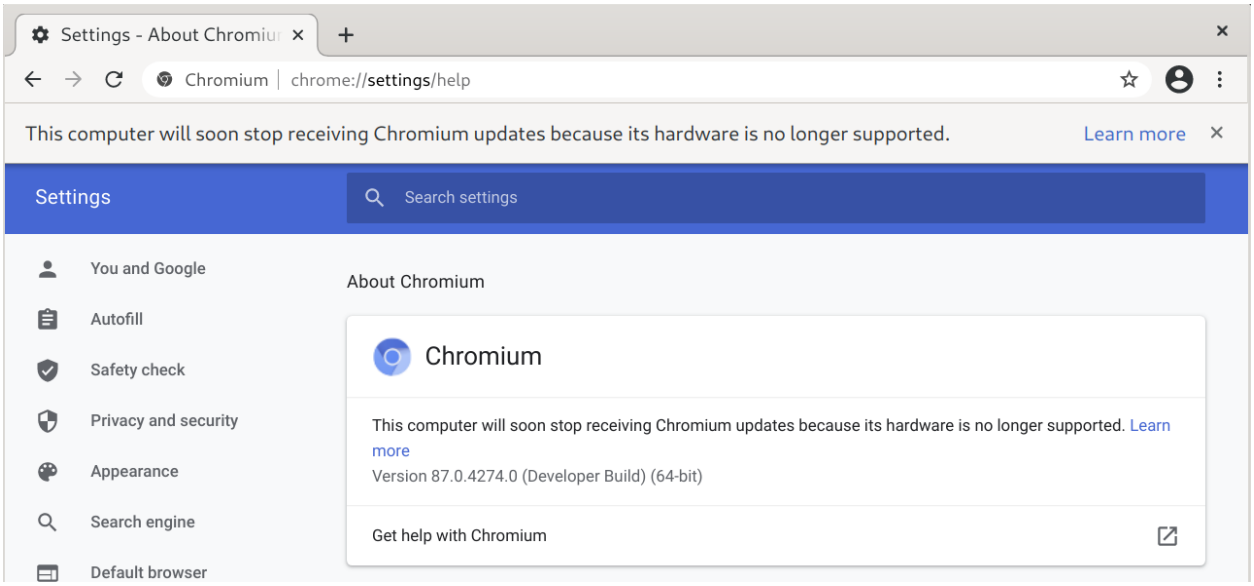
- Download and install Google Chrome (the *System requirements to use Chrome* section)
- Chrome Browser system requirements

The serving infrastructure for the Chrome Updater will need to be updated to not serve upgrades >= M89 to x86 users without SSE3 support.

Warning about the upcoming SSE3 requirement

Until we require SSE3, Chrome will warn impacted users (with x86 CPUs that don't support SSE3) that their computers will soon be unsupported.

The implementation will use the framework in //chrome/browser/obsolete_system. This will result in a dismissable warning bar, and a permanent warning in the `chrome://settings/help` page. The results are illustrated in the prototype below.



The code change is prototyped in https://crrev.com/c/2427631.

Expected impact on usage

Chrome will no longer be usable on computers with x86 processors that support SSE2 but do not support SSE3. This is expected to reduce Chrome usage on Windows by a very small amount.

Historical notes

- SSE2 was introduced on Intel x86 CPUs in 2000, and on AMD x86 CPUs in 2003. It is supported by all 64-bit CPUs.
- SSE3 was introduced on Intel CPUs in 2003, and on AMD CPUs in 2005.
- Chrome started requiring SSE2 in 2014.

Compiler notes

- Clang and GCC accept target CPU specifications via an assortment of arguments, which include the fundamental `-march`, `-mcpu`, and `-mtune`, and synonyms like `-msse3`. These impact which instructions are generated, both by the code generator and directly via intrinsics. Attempting to use intrinsics that are not supported by the current architecture results in compilation errors. Clang and GCC support targeting all Intel and AMD microarchitectures, so they will generate SSE3 code after this change.
- MS Visual Studio's C++ compiler (MSVC) accepts target CPU specifications via an `/arch:` flag. The flag only impacts the code generator's output. All intrinsics can be used, no matter what `/arch:` flag gets passed. The supported `/arch:` values for 64-bit CPUs are SSE2 (default), AVX, AVX2, and AVX512. This means MSVC will only generate SSE3 code when explicitly instructed to do so.

Windows

Our analysis (Googler-only, sorry) indicates that there is a very small number of Windows devices running Chrome with x86 processors that do not support SSE3.

The Windows versions supported by Chrome require SSE2, but not SSE3.

- Windows 7 removed support for x86 processors without SSE2 support in an update.
- Windows 8 for x86 processors requires SSE2 support.
- Windows 10 for x86 processors requires SSE2 support.

Chrome for Windows started requiring SSE2 in M35, which was pushed to Stable in May, 2014.

Android

This change is not expected to reduce usage on Android.

Chrome currently requires Android L or later. Out of all Intel-based Android phones running Android L or above, the earliest processor used is Atom Z2520, which supports SSSE3.

Chrome OS

This change is not expected to reduce usage on Chrome OS.
Historical notes.

- In Nov 2011, Chrome OS started assuming SSSE3 on x86 (`-march=atom` implies SSSE3).
- In March 2012, the x86 requirement was reduced from SSSE3 support to SSE3.
- The venerable Cr-48 had an Atom N455 processor, which supports SSSE3.
- (SSSE3 is newer than SSE3, ignoring the confusing naming.)

Out of the Chrome OS devices with AUE (Auto Update Expiration) dates in the future, the oldest CPUs that are still supported are Silvermont (Baytrail / Braswell), which can do SSE4.2, and Broadwell, which can do AVX.

From a different angle, as of M63, the Zip Archiver code assumes SSE4.1 support on x86.

macOS

This change will be a no-op on macOS.

Chrome only ships 64-bit builds on macOS. Our build configuration currently uses clang's default CPU settings on macOS. When targeting 64-bit systems, this default is `-march=core2`, which includes SSSE3 support.

Future changes will be based on Chrome's minimum supported macOS version, combined with the hardware supported by that macOS version. From that perspective, Chrome currently requires macOS 10.10 or later. The computers supported by macOS 10.10 have Intel Core 2 Duo processors or better, which support SSSE3. Some Core 2 Duo processors support SSE4.1, but older Mac models supported by 10.10 ship with processors that only support SSSE3.

Alternatives

Dynamic dispatch

Dynamic dispatch makes it possible to take advantage of performance improvements in a recent microarchitecture, like SSE3, while still supporting older microarchitectures, like SSE2. Upgrading the minimum required microarchitecture will reduce, but not eliminate, the need to use dynamic dispatch.

For reasons that will be summarized below, dynamic dispatch is expensive to implement, so it is only used to add fast paths for very high-benefit cases like BoringSSL, libaom, and Skia. Small opportunities are missed.

Dynamic dispatch increases binary size, as we ship multiple compiled versions of the same code. The binary size increase is mitigated by the fact that dynamic dispatch is used rarely, due to the engineering cost.

At a high level, we ship binary code compiled for the baseline microarchitecture (SSE2) together with optimized variants that target recent microarchitectures (for example, SSE3, SSSE3, and AVX). At runtime, we detect the CPU's microarchitecture and we do dynamic dispatch (execute the most suitable variant). The main engineering subtlety here is ensuring that we don't crash Chrome by executing code targeting a recent microarchitecture (SSE3) on an older machine (only supporting SSE2).

The rest of this section provides background on the general techniques employed in dynamic dispatch. The content is not specific to the change proposed here.

Dynamic dispatch structure

The first step towards taming the complexity is isolating the fast path code to its own compilation unit (.c / .cc file) plus header.

1. Each microarchitecture variant gets its own copy of the compilation unit, for example `swirl_baseline.cc`, `swirl_sse3.cc`, `swirl_avx.cc`.
2. The microarchitecture-specific compilation unit defines an entry point to the fast path, for example `SwirlBaseline()`, `SwirlSse3()`, and `SwirlAvx()`.

3. The entry points above are declared in per-microarchitecture header files (`swirl_baseline.h`, `swirl_sse3.h`, `swirl_avx.h`) or all in one file (`swirl_variants.h`),
4. Each microarchitecture-specific compilation unit ends up in its own source set with corresponding compilation options, such as `-msse3` or `-mavx`.
5. Often, the microarchitecture-specific compilation units use intrinsics directly to ensure that the compiler outputs the desired optimized code.

The dynamic dispatch code (in our example, `Swirl()` in `swirl.cc`) queries the CPU's microarchitecture, and calls the most appropriate function between `SwirlBaseline()`, `SwirlSse3()`, and `SwirlAvx()`. The dynamic dispatch code is in a compilation unit with the baseline compilation options (no `-msse3` or `-mavx`).

Preventing ODR violations

The main problem in this setup is avoiding breaking ODR (the C++ One Definition Rule). To accomplish this, the microarchitecture-specific compilation units cannot include any header that defines an `inline` function. This includes many standard library files like `<vector>`, because all template functions are `inline`.

Normally, it's OK to have an `inline` function included in multiple compilation units. The C++ language has a special provision that multiple definitions of `inline` functions don't count as ODR, as long as they are token-for-token identical. However, compiling the same code with different code generation options (such as target microarchitectures) effectively counts as having two different definitions.

Getting this wrong can lead to bugs that are really difficult to track down! At the time of this writing, linkers will assume that the definitions of `inline` functions are the same across all compilation units, and just pick one. This could, for example, cause all uses of `<vector>` in Chrome to execute AVX2 instructions.

Preventing ABI differences

Microarchitecture-specific compilation options, like `-mavx`, can cause subtle changes in the ABI (application binary interface) used by the compiler for a compilation unit. The interface between the dispatch code and the microarchitecture-specific code may be impacted by these ABI changes, as shown in the example below. Fortunately, bugs caused by ABI differences are easy to detect by good automated test coverage.

For example, the declaration for `SwirlAvx()` might be

```
float SwirlAvx(float input[8]);
```

In the dynamic dispatch code `swirl.cc`, the ABI used by the compiler may dictate that `input` is passed using two SSE registers. However, in the AVX-specific `swirl_avx.cc` , the

ABI may dictate that `input` is passed using one AVX register. When `SwirlAvx()` is called, the parameter won't be passed correctly, leading to buggy behavior.

Freeze Chrome x86 requirements at SSE2

History suggests that new x86 microarchitectures will be released regularly. This alternative would result in an ever-increasing gap between the SSE2 microarchitecture used by default and the unrealized potential of modern processors. This gap would add pressure to use dynamic dispatch in more places, and to use more microarchitecture-specific variants at each dynamic dispatch site.

Increase Chrome x86 requirements directly to SSSE3

Jumping to SSSE3 would let us enable a fast path in snappy, and perhaps other libraries. This jump would also allow us to remove a dynamic dispatch alternative in each of the following third-party libraries: BoringSSL, dav1d, libaom, libvpx, PDFium, Skia.

This alternative was rejected, and we prefer to get to SSSE3 in two smaller steps, instead of one big step. The main reason is that Chrome runs on a small (but non-trivial) number of x86 computers that support SSE3 but not SSSE3.

Warn users with unsupported x86 CPUs

The Chrome installer for Windows attempts to exit early on an unsupported CPU. The word *attempts* is used to mean that there are no guarantees that the installer won't attempt to execute an unsupported instruction (such as SSE2) given the GN build configuration.

If the engineering effort would be deemed worthwhile, it would be possible to use a subset of the Dynamic dispatch techniques described earlier to give a clear error on unsupported CPUs. Specifically, we would carve out some code that gets compiled for a different microarchitecture, but in this case the target would be lower than the baseline. There would be no dynamic dispatch, as the code would be safe to call everywhere.

At a high level, the changes required are below.

- Carve out a subset of base::CPU that can be compiled without the standard library, for use with the mini installer.
- Carve out a bit of startup code around `//chrome/installer/mini_installer`'s `MainEntryPoint()` (`WinMain()` equivalent) that checks for the desired SSE support on x86 CPUs and calls `MessageBoxW()` if the requirements aren't met. This startup code would be compiled targeting the lowest microarchitecture that makes sense. The research in the *Expected impact on usage* section suggests this is SSE2.
- Carve out a bit of startup code around `//chrome/installer/setup`'s `wWinMain()`, with the same approach as above.

Since this is an alternative, we haven't figured out the internationalization story for the message in the modal dialog. TBD if supporting translated resources here is worthwhile, or if we want to rely on folks being able to search for an error message on the web.

Metrics

We propose considering this change equivalent to a toolchain upgrade.

Our analysis (Googler-only, sorry) indicates that there is a very small number of Windows devices running Chrome with x86 processors that do not support SSE3. The *Expected impact on usage* section has more details.

Success metrics

No explicit metrics. Having the change stick in the tree will be considered success.

Regression metrics

Crash rates.

Experiments

No experiments.

A/B binary tests carry a non-trivial engineering cost, which are not warranted for this change.

Rollout plan

Waterfall, with a bit of care around the timing.

We propose landing the change requiring SSE3 in M89. The change will be landed early in the M89 cycle, to maximize exposure before the change reaches Stable.

The warning about missing SSE3 support will be added to Chrome as soon as possible. Update: The warning was landed in time for M87.

Core principle considerations

Speed

This change appears to result in a tiny binary size saving (5kb on a Chrome official build).

In general, giving the compiler more freedom in CPU instruction selection results in same-or-better speed. There is one edge case, summarized below, which will become relevant when we consider upgrading to AVX. This edge case is not relevant for this particular proposal of upgrading from SSE2 to SSE3.

SIMD (Single instruction, multiple data) instructions on x86 processors draw more power than regular instructions. In some cases, the extra power consumption increases the CPU's temperature, which causes the CPU to transition to a lower frequency. After expensive SIMD instruction use ceases, the CPU eventually cools down and transitions back to a higher frequency.

The power draw is generally proportional to the register size. On Intel processors, this effect was observed when executing AVX (Advanced Vector Extensions) instructions, which operate on 256-bit registers. Another stage of this effect occurs when executing AVX-512 instructions, which (unsurprisingly) operate on 512-bit registers. All cores of a CPU are hit with a frequency reduction, even if a single core executes AVX instructions.

This effect is particularly visible in workloads that consist mostly of 32/64-bit instructions and SSE instructions, where a human or automated optimizer inserts a few AVX or AVX-512 instructions. The "optimizations" can have a negative overall impact, as the slowdown induced by the CPU frequency drop may significantly outweigh the speedup from a few AVX instructions.

No significant effort was observed when executing SSE instructions, which operate on 128-bit registers. So, from this perspective, speed should not be an issue as we upgrade Chrome throughout SSE revisions.

Google-internal performance wisdom (Googlers-only, sorry!) matches the public findings linked above.

Security

We propose treating this change as a toolchain upgrade. No changes to Chrome's security profile are expected.

Privacy considerations

N/A.

Testing plan

Requiring SSE3

No special plan. The main concern is the (tiny) possibility of compiler bugs exposed by the new build configuration. We assume that the existing automated and manual testing will catch any regressions. In other words, if a feature isn't tested, it's already broken.

Warning about the upcoming SSE3 requirement

The //chrome/browser/obsolete_system integration will be tested manually before the code is landed. The CPU detection logic cannot be tested by automated tests or by QA engineers because getting an x86 CPU without SSE3 support is quite difficult at the time of this writing.

The manual testing process will follow the steps below on Windows, macOS, and Linux.

1. Build and launch Chrome. Confirm that no warnings are shown.
2. Stub the CPU check (in IsObsoleteCpu()) to always return true.
3. Rebuild Chrome.
4. Launch Chrome. Confirm the existence of an SSE3 warning in a bar.
5. Visit `chrome://settings/help`. Confirm the existence of an SSE3 warning.

Followup work

The warning about missing SSE3 support will need to be removed after Chrome starts requiring SSE3. At that point, x86 machines without SSE3 support will not be able to start Chrome, so the warning code will be unreachable.

Third-party libraries may rely on build configuration to enable SSE3-optimized paths. It might be possible to improve or simplify the build configuration for these libraries. For example, libaom currently builds a libaom_intrinsics_sse2 target that would no longer be necessary.

After this change sticks, we can think about the timelines for requiring SSSE3 on x86 processors. (This is not a typo. SSSE3 stands for *Supplemental* Streaming SIMD Extensions 3, and is the next step after SSE3.) This will require coordination with Chrome OS, as the Chromium OS codebase currently requires SSE3.